Neural Networks are Function Approximation Algorithms

By Jason Brownlee

Supervised learning in machine learning can be described in terms of function approximation.

Given a dataset comprised of inputs and outputs, we assume that there is an unknown underlying function that is consistent in mapping inputs to outputs in the target domain and resulted in the dataset. We then use supervised learning algorithms to approximate this function.

Neural networks are an example of a supervised machine learning algorithm that is perhaps best understood in the context of function approximation. This can be demonstrated with examples of neural networks approximating simple one-dimensional functions that aid in developing the intuition for what is being learned by the model.

In this tutorial, you will discover the intuition behind neural networks as function approximation algorithms.

After completing this tutorial, you will know:

- Training a neural network on data approximates the unknown underlying mapping function from inputs to outputs.
- One dimensional input and output datasets provide a useful basis for developing the intuitions for function approximation.
- How to develop and evaluate a small neural network for function approximation.

Let’s get started.

Neural Networks are Function Approximation Algorithms
Photo by daveynin, some rights reserved.

Tutorial Overview

This tutorial is divided into three parts; they are:

1. What Is Function Approximation
2. Definition of a Simple Function
3. Approximating a Simple Function

What Is Function Approximation

Function approximation is a technique for estimating an unknown underlying function using historical or available observations from the domain.

Artificial neural networks learn to approximate a function.

In supervised learning, a dataset is comprised of inputs and outputs, and the supervised learning algorithm learns how to best map examples of inputs to examples of outputs.

We can think of this mapping as being governed by a mathematical function, called the **mapping function**, and it is this function that a supervised learning algorithm seeks to best approximate.

Neural networks are an example of a supervised learning algorithm and seek to approximate the function represented by your data. This is achieved by calculating the error between the predicted outputs and the expected outputs and minimizing this error during the training process.

It is best to think of feedforward networks as function approximation machines that are designed to achieve statistical generalization, occasionally drawing some insights from what we know about the brain, rather than as models of brain function.

— Page 169, Deep Learning, 2016.

We say "*approximate*" because although we suspect such a mapping function exists, we don't know anything about it.

The true function that maps inputs to outputs is unknown and is often referred to as the **target function**. It is the target of the learning process, the function we are trying to approximate using only the data that is available. If we knew the target function, we would not need to approximate it, i.e. we would not need a supervised machine learning algorithm. Therefore, function approximation is only a useful tool when the underlying target mapping function is unknown.

All we have are observations from the domain that contain examples of inputs and outputs. This implies things about the size and quality of the data; for example:

- The more examples we have, the more we might be able to figure out about the mapping function.
- The less noise we have in observations, the more crisp approximation we can make of the mapping function.

So why do we like using neural networks for function approximation?

The reason is that they are a **universal approximator**. In theory, they can be used to approximate any function.

… the universal approximation theorem states that a feedforward network with a linear output layer and at least one hidden layer with any "squashing" activation function (such as the logistic sigmoid activation function) can approximate any […] function from one finite-dimensional space to another with any desired non-zero amount of error, provided that the network is given enough hidden units

— Page 198, Deep Learning, 2016.

Regression predictive modeling involves predicting a numerical quantity given inputs. Classification predictive modeling involves predicting a class label given inputs.

Both of these predictive modeling problems can be seen as examples of function approximation.

To make this concrete, we can review a worked example.

In the next section, let’s define a simple function that we can later approximate.

Definition of a Simple Function

We can define a simple function with one numerical input variable and one numerical output variable and use this as the basis for understanding neural networks for function approximation.

We can define a domain of numbers as our input, such as floating-point values from -50 to 50.

We can then select a mathematical operation to apply to the inputs to get the output values. The selected mathematical operation will be the mapping function, and because we are choosing it, we will know what it is. In practice, this is not the case and is the reason why we would use a supervised learning algorithm like a neural network to learn or discover the mapping function.

In this case, we will use the square of the input as the mapping function, defined as:

- y = x^2

Where *y* is the output variable and *x* is the input variable.

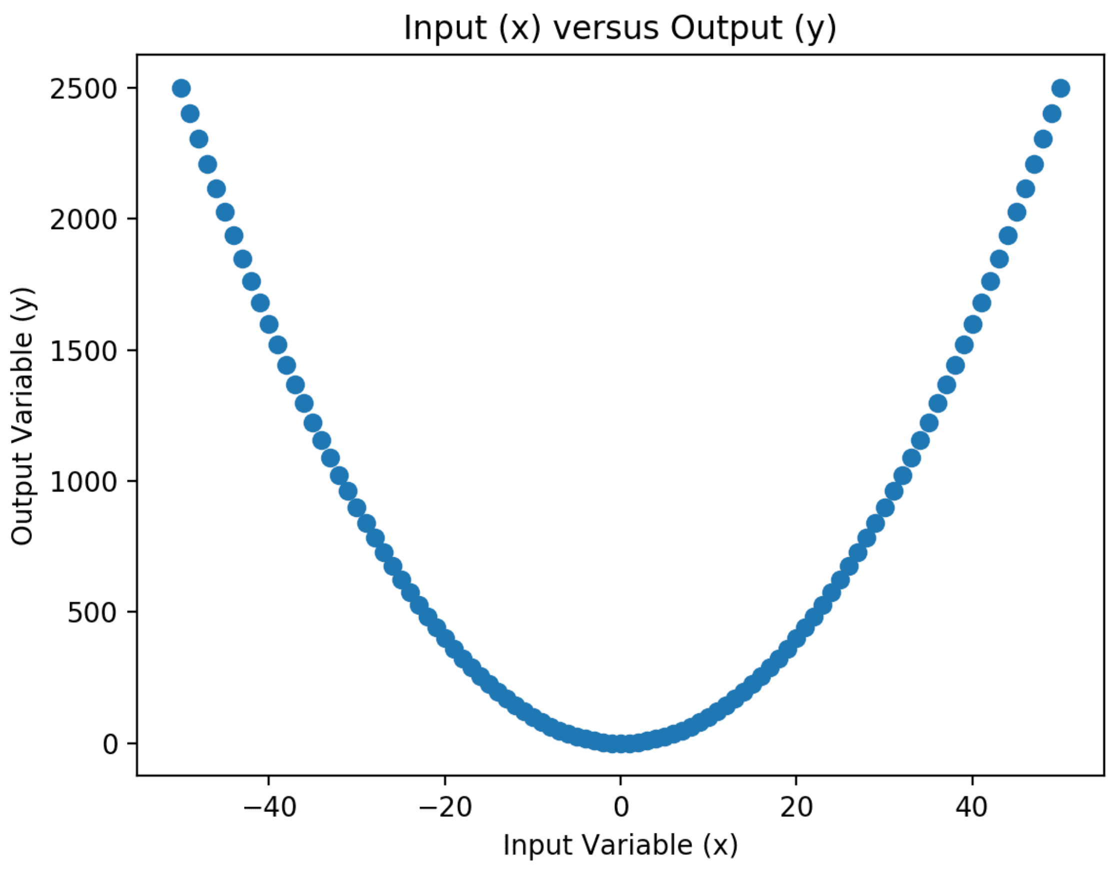
We can develop an intuition for this mapping function by enumerating the values in the range of our input variable and calculating the output value for each input and plotting the result.

The example below implements this in Python.

```
# example of creating a univariate dataset with a given mapping function
from matplotlib import pyplot
# define the input data
x = [i for i in range(-50,51)]
# define the output data
y = [i**2.0 for i in x]
# plot the input versus the output
pyplot.scatter(x,y)
pyplot.title('Input (x) versus Output (y)')
pyplot.xlabel('Input Variable (x)')
pyplot.ylabel('Output Variable (y)')
pyplot.show()
```

Running the example first creates a list of integer values across the entire input domain.

The output values are then calculated using the mapping function, then a plot is created with the input values on the x-axis and the output values on the y-axis.



Scatter Plot of Input and Output Values for the Chosen Mapping Function

The input and output variables represent our dataset.

Next, we can then pretend to forget that we know what the mapping function is and use a neural network to re-learn or re-discover the mapping function.

Approximating a Simple Function

We can fit a neural network model on examples of inputs and outputs and see if the model can learn the mapping function.

This is a very simple mapping function, so we would expect a small neural network could learn it quickly.

We will define the network using the Keras deep learning library and use some data preparation tools from the scikit-learn library.

First, let's define the dataset.

```
...
# define the dataset
x = asarray([i for i in range(-50,51)])
y = asarray([i**2.0 for i in x])
print(x.min(), x.max(), y.min(), y.max())
```

Next, we can reshape the data so that the input and output variables are columns with one observation per row, as is expected when using supervised learning models.

```
...
# reshape arrays into into rows and cols
x = x.reshape((len(x), 1))
y = y.reshape((len(y), 1))
```

Next, we will need to scale the inputs and the outputs.

The inputs will have a range between -50 and 50, whereas the outputs will have a range between -50^2 (2500) and 0^2 (0). Large input and output values can make training neural networks unstable, therefore, it is a good idea to scale data first.

We can use the MinMaxScaler to separately normalize the input values and the output values to values in the range between 0 and 1.

```
...
# separately scale the input and output variables
scale_x = MinMaxScaler()
x = scale_x.fit_transform(x)
scale_y = MinMaxScaler()
y = scale_y.fit_transform(y)
print(x.min(), x.max(), y.min(), y.max())
```

We can now define a neural network model.

With some trial and error, I chose a model with two hidden layers and 10 nodes in each layer. Perhaps experiment with other configurations to see if you can do better.

```
...
# design the neural network model
model = Sequential()
model.add(Dense(10, input_dim=1, activation='relu', kernel_initializer='he_uniform'))
model.add(Dense(10, activation='relu', kernel_initializer='he_uniform'))
model.add(Dense(1))
```

We will fit the model using a mean squared loss and use the efficient adam version of stochastic gradient descent to optimize the model.

This means the model will seek to minimize the mean squared error between the predictions made and the expected output values (*y*) while it tries to approximate the mapping function.

```
...
# define the loss function and optimization algorithm
model.compile(loss='mse', optimizer='adam')
```

We don’t have a lot of data (e.g. about 100 rows), so we will fit the model for 500 epochs and use a small batch size of 10.

Again, these values were found after a little trial and error; try different values and see if you can do better.

```
...
# ft the model on the training dataset
model.fit(x, y, epochs=500, batch_size=10, verbose=0)
```

Once fit, we can evaluate the model.

We will make a prediction for each example in the dataset and calculate the error. A perfect approximation would be 0.0. This is not possible in general because of noise in the observations, incomplete data, and complexity of the unknown underlying mapping function.

In this case, it is possible because we have all observations, there is no noise in the data, and the underlying function is not complex.

First, we can make the prediction.

```
...
# make predictions for the input data
yhat = model.predict(x)
```

We then must invert the scaling that we performed.

This is so the error is reported in the original units of the target variable.

```
...
# inverse transforms
x_plot = scale_x.inverse_transform(x)
y_plot = scale_y.inverse_transform(y)
yhat_plot = scale_y.inverse_transform(yhat)
```

We can then calculate and report the prediction error in the original units of the target variable.

```
...
# report model error
print('MSE: %.3f' % mean_squared_error(y_plot, yhat_plot))
```

Finally, we can create a scatter plot of the real mapping of inputs to outputs and compare it to the mapping of inputs to the predicted outputs and see what the approximation of the mapping function looks like spatially.

This is helpful for developing the intuition behind what neural networks are learning.

```
...
# plot x vs yhat
pyplot.scatter(x_plot,yhat_plot, label='Predicted')
pyplot.title('Input (x) versus Output (y)')
pyplot.xlabel('Input Variable (x)')
pyplot.ylabel('Output Variable (y)')
pyplot.legend()
pyplot.show()
```

Tying this together, the complete example is listed below.

```
# example of fitting a neural net on x vs x^2
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import mean_squared_error
from keras.models import Sequential
from keras.layers import Dense
from numpy import asarray
from matplotlib import pyplot
# define the dataset
x = asarray([i for i in range(-50,51)])
y = asarray([i**2.0 for i in x])
print(x.min(), x.max(), y.min(), y.max())
# reshape arrays into into rows and cols
x = x.reshape((len(x), 1))
y = y.reshape((len(y), 1))
# separately scale the input and output variables
scale_x = MinMaxScaler()
x = scale_x.fit_transform(x)
scale_y = MinMaxScaler()
y = scale_y.fit_transform(y)
print(x.min(), x.max(), y.min(), y.max())
# design the neural network model
model = Sequential()
model.add(Dense(10, input_dim=1, activation='relu', kernel_initializer='he_uniform'))
model.add(Dense(10, activation='relu', kernel_initializer='he_uniform'))
model.add(Dense(1))
# define the loss function and optimization algorithm
model.compile(loss='mse', optimizer='adam')
# ft the model on the training dataset
model.fit(x, y, epochs=500, batch_size=10, verbose=0)
# make predictions for the input data
yhat = model.predict(x)
# inverse transforms
x_plot = scale_x.inverse_transform(x)
y_plot = scale_y.inverse_transform(y)
yhat_plot = scale_y.inverse_transform(yhat)
# report model error
print('MSE: %.3f' % mean_squared_error(y_plot, yhat_plot))
```

```
# plot x vs y
pyplot.scatter(x_plot,y_plot, label='Actual')
# plot x vs yhat
pyplot.scatter(x_plot,yhat_plot, label='Predicted')
pyplot.title('Input (x) versus Output (y)')
pyplot.xlabel('Input Variable (x)')
pyplot.ylabel('Output Variable (y)')
pyplot.legend()
pyplot.show()
```

Running the example first reports the range of values for the input and output variables, then the range of the same variables after scaling. This confirms that the scaling operation was performed as we expected.

The model is then fit and evaluated on the dataset.

Note: Your results may vary given the stochastic nature of the algorithm or evaluation procedure, or differences in numerical precision. Consider running the example a few times and compare the average outcome.

In this case, we can see that the mean squared error is about 1,300, in squared units. If we calculate the square root, this gives us the root mean squared error (RMSE) in the original units. We can see that the average error is about 36 units, which is fine, but not great.

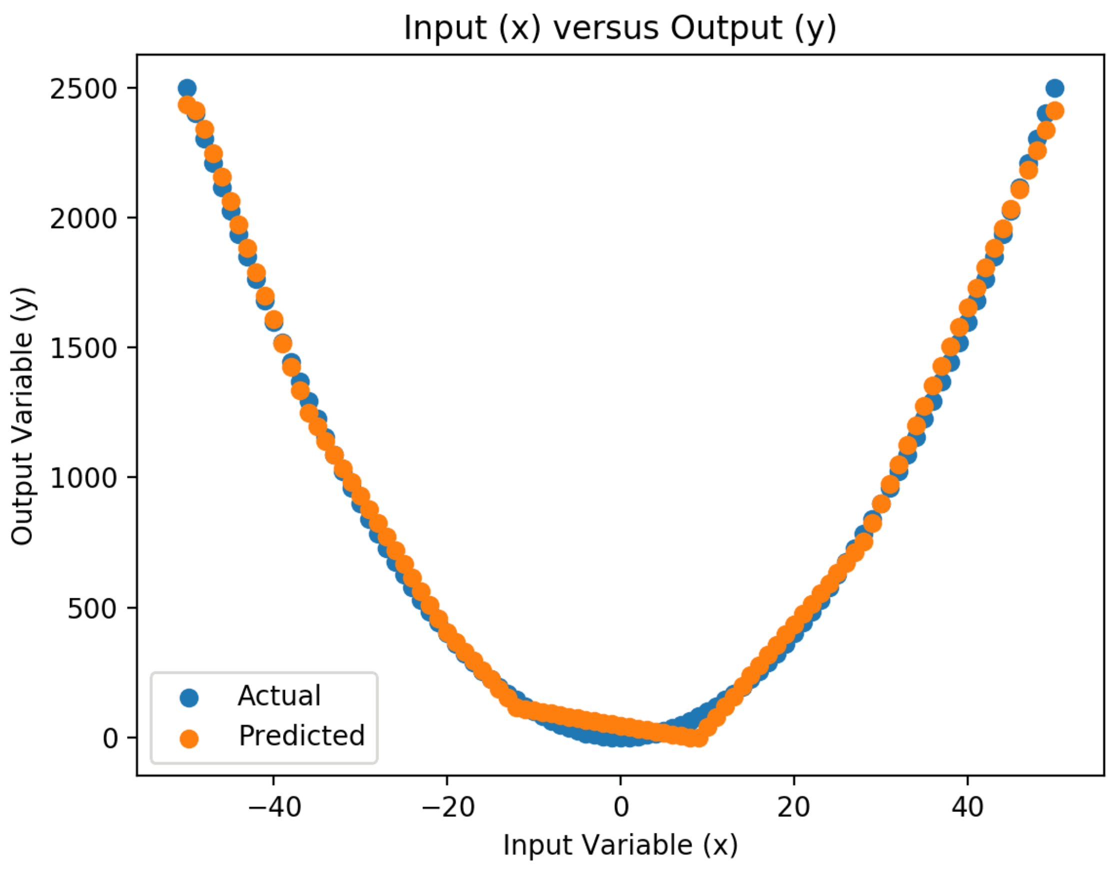
What results did you get? Can you do better?
Let me know in the comments below.

```
-50 50 0.0 2500.0
0.0 1.0 0.0 1.0
MSE: 1300.776
```

A scatter plot is then created comparing the inputs versus the real outputs, and the inputs versus the predicted outputs.

The difference between these two data series is the error in the approximation of the mapping function. We can see that the approximation is reasonable; it captures the general shape. We can see that there are errors, especially around the 0 input values.

This suggests that there is plenty of room for improvement, such as using a different activation function or different network architecture to better approximate the mapping function.



Scatter Plot of Input vs. Actual and Predicted Values for the Neural Net Approximation

Further Reading

This section provides more resources on the topic if you are looking to go deeper.

Tutorials

- Your First Deep Learning Project in Python with Keras Step-By-Step

Books

- Deep Learning, 2016.

Articles

- Function approximation, Wikipedia.